Gmail by Google

Harshit Sanghvi <sanghvi.harshit@gmail.com>

Programming Assignment -- PLEASE READ TO THE END

John Viega <viega@nyu.edu> Mon, Mar 31, 2014 at 3:33 PM
To: viega@nyu.edu

All,

Today you will be getting grades back from programming assignment 1. Some of you did well, and some of you did poorly.

Please read to the end of this message, as there are going to be some changes around future programming assignments.

Those of you who had problems generally had the same problems:

1) You didn't do enough testing. The specification was pretty thorough, and every test case was derived from what was in the specification — if something was left ambiguous, I did not test it.
2) In fact, many of you didn't even bother to test against the six cases I provided you. It was shocking how many people only had SOME of those test cases working.
3) You didn't follow basic directions. A lot of people didn't have their makefiles generate the right executables, didn't read from the correct input file, etc. This time, I tweaked your programs for you so that they would run in my test bed, but I will NOT do it again.

None of my test cases were large— nothing had more than two dozen tokens, and most had fewer than 10.

The six cases you already got were worth 5 points each. The other 54 cases were worth two points each. That means there were 138 total points. Instead of scaling it down to 100, your grade is how many points you got. Yes, people got scores in the 100+ range (the highest grade was a 114).

Here's how your score translates to a letter grade:
97-138 A+
93-96 A
90-92 A-
87-89 B+
83-86 B
80-82 B-
77-79 C+
73-76 C
70-72 C-
67-69 D+
40-66 D
25-39 D-
Below 25: F

I bent over backwards to give people points, even when they did not follow directions. For instance, some people didn't follow guidelines for white space, and had tons of extra white space. I ignored such problems this time, but will not do so again.

Two people submitted programs that did not build or produce output. Those people got 0s. For everyone else, you will see whether you passed or failed each of the 60 test cases.

There was only one test case where nobody got the correct answer.
It was Good 2:

```
global {
   somevariable = "Foo\rbar"
};
```

I'm a bit surprised nobody got this, because people did get a similar test case where a newline was inserted. And being able to handle this case was mentioned quite clearly in the spec when I talked about quoted strings.

I'll give you three more of my test cases— these three only had one person get the case right each:

First is Good 19:
```
global {}
host # This is a comment
foo
{
 x=1
}
```

Expected output:
```
GLOBAL:
HOST foo:
    I::x:1
```

The problem with this one is that people needed to expect that comments can come anywhere, except in a quoted string. And, actually, they can't really happen in the middle of an assignment, either, since it would make the assignment illegal— assignments can span multiple lines. The comment itself would be fine, it's just that the assignment would necessarily be malformed.

The second problem spot was Bad 11:
```
global {
 x = 4.0
};;
```

Expected output for this one:
```
GLOBAL:
    F::x:4.0
ERR:P:3
```

For this test case, you had to follow the spec's guidance on semicolons. It says there may be AN optional semicolon — not multiple semicolons.

That wasn't the main problem here. A couple of people got the semicolons right, but only one ALSO printed out what had matched so far when the error was found, which is clearly stated in the spec as something you had to do. Two total test cases tested that printing functionality.

Note that this error is something that really should only be found in the parser, not the lexer. But in general, I IGNORED where you found the error.

The third problem spot was Bad 14:
```
global {
 x = 10
}
host this {
  x = "Bad string\"
}
```

Which should have output either this:

```
ERR:L:5
```

Or, if you were lexing as you parsed, you would have been expected to print out this:

```
GLOBAL:
    I::x:10
ERR:L:5
```

Both are correct interpretations of the specification, and both got you credit (note that the bad token appears on line 5— not the last line of the file). Again, even if you mistakenly had a P instead of an L, you got credit.

In the next round of grading, I am going to rename stuff in the test cases I've given you, but otherwise all TEN of them will still be in the test suite, along with the other 50. I will not change the weighting of cases. I may add a few more test cases, but if so, this will only increase your potential to get points (the grading scale will be the same).

I recommend you do what I do: go through each line of the spec and ask yourself, "how can I test this?" and come up with a few very simple test cases. See if your output matches what you would expect. Test everything — where comments can go, where whitespace makes sense and where it doesn't, where you can

put the word global (there's nothing restricting it from being a variable, for instance), what happens if you leave stuff out, whether you handle CASE properly, ...

Thorough testing is absolutely critical to software engineering. If you can't produce reasonably robust code, you won't be respected for your programming skills, and if you don't test pretty thoroughly, you won't generally end up with robust code.

Also, I want you to recognize that being able to read a simple configuration file into a data structure is something that is a fundamentally basic programming task. However, I do realize that some of your are still only novice programmers, and this problem is compounded when learning new languages and technologies.

Therefore, I've decided that, to get the most out of the class, I'm going to add a fourth part to this assignment, but take away the second project. So there's now only one project, but it has FOUR parts. For the fourth part, you are going to do this exact assignment in any programming language you choose.

I think this will help you figure out HOW to parse a configuration file if you do it in a language where you're already comfortable programming. If you can focus on doing a good job in your own preferred programming language, then you can just worry about translating what you did into other paradigms.

For whatever language you choose, you must meet the same basic requirements as for other languages in terms of compiling and running. Most importantly, when I run your program, you must look in test.cfg in the current directory, read that in, and then output to standard output (via print or similar).

If you're choosing a compiled language, have your Makefile create the executable: imp4
If you're choosing an interpreted language, have your main program called imp4, with the appropriate file extension. For instance, if you're using Python, use imp4.py

I will be running everything in the same test environment as before. You need to make sure your program will work in my test environment. If you choose any of the following programming languages, here's info on the distribution I use to test (again, on an up-to-date Ubuntu box— let me know if you need access):

Python 2 — Comes standard w/ Ubuntu— /usr/bin/python
Python 3 — Comes standard w/ Ubuntu — /usr/bin//python3
Perl — Comes standard w/ Ubuntu — /usr/bin/perl
Ruby — Standard distro; on Ubuntu run apt-get install ruby, then lives in /usr/bin/ruby
Java — apt-get install openjdk-7-jdk (usr/bin/javac)
C — use the local cc compiler (/usr/bin/cc)
C++ — use the local c++ compiler (/usr/bin/c++)
JavaScript — you will have to make this work from the command line using the Rhino JavaScript engine. Do: apt-get install rhino, then you will have /usr/bin/js

If there's some other language that you'd like to use, talk to me before you begin. I'll make sure I can run it first, and then will let you know how I'm going to run it in my environment. I'll make a reasonable effort to accommodate whatever language you want.

As before, you are to use no outside libraries or parser generators. You can use libraries that come standard with the language, though.

If you choose C or C++, that's fine, just note that it has to be all your code— I will be making sure that you're not using any of the code that Flex or Yacc will generate for you, and will give you a 0 if you do.

Here's the schedule for the programming assignments:

April 13, before MIDNIGHT: Turn in your second program, preferably imp4
April 27, before MIDNIGHT: Turn in your third program, preferably imp2 (or imp1 for those who already did imp2)
May 11, before MIDNIGHT: Turn in your final program, preferably imp3

There will be NO further extensions. Late assignments will get an automatic 0.

Additionally, for those who wish to re-submit assignment 1, you may do so at any point before May 12, and I will blend your two grades— the regrade will count twice as much as the first grade. For example, if you got a 0 on the original, but get a perfect score (138) on the regrade, you would get: (0 + 138*2)/3 = 92.

There will be NO resubmissions for the second, third and fourth programs. Your grade will be your grade.

From this point forward, there will be no flexibility if you do not follow my instructions carefully. If you have problems with whitespace, with filenames, with executable names, then you are at risk of all your hard work resulting in a zero.

All four assignments will be evenly weighted, and they will total 35% of your semester grade. As before, homework will be 15%, the midterm 20%, and the final 30%. I am not sure if I will have the midterm graded before Friday's class, but we will review the contents.

- Prof. Viega

p.s., for test cases 5 and 6, I was more liberal on these, because the example output assumed that you weren't going to print unless you'd completed parsing a group, but some people would output even a partial group. Both are okay vs. the spec. So for case 5, as an example, either of these output files are good:

```
ERR:P:7
```

Or:

```
GLOBAL:
    I::port:80
    I::num_threads:4
    S::ssl_key:/etc/mycreds.pem
ERR:P:7
```

And, again, either a P or an L (parser error or lexer error) is acceptable in this context.